EN.540.635 “Software Carpentry”

Python, a Crash Course

Introduction

Here we will go through the fundamentals of Python, alongside explanatory “google” searching. Using Google to find answers to questions you might have about specific programming techniques is a very powerful tool! To start off, Python itself is known as a high-level programming langauge that was created by Guido van Rossum in 1991. The current stable release version of Python is Python 3; previous versions of Python have been officially discontinued as of the start of 2020, so you should make sure that the version of Python you are using is Python 3 or greater.

Python Interface

Python is a *scripting language*, and as such it can be run directly from a command line. If installed on the computer, the Python interface can be called from the command line by typing the word python (with no other characters) as follows:

```
~ python
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

From here, we could start coding! However, what if we wanted to write a long, complex code of some kind? Unfortunately if we make an error while typing it into the command line we need to start over again. This isn’t very useful to the programmer. However, Python makes life easier by letting you type all the code ahead of time in a separate file, and then we can run our code using the *interpreter* to parse through every line of code in the file. Below is an example of such a file - where we have code that will convert Cartesian coordinates to polar coordinates (NOTE! we do not expect you to already know everything going on below, but we have touched on all the elements in the code so you should be able to read it at least):

```
# Import the math library so we have functions like cos, sin, sqrt, and
# constants like pi
import math


def cartesian_to_polar(x, y):
    # NOTE! This code only works for x > 0 and y > 0

    # If x == y == 0, this is a weird boundary case we accomodate for
    if x == y and y == 0:
        return 0, 0

    # Get the polar radial distance
    r = math.sqrt(x ** 2 + y ** 2)
    # Get the polar angle
    # NOTE! We need to accomodate for x = 0, as we will be dividing
    if x == 0:
        # It should be either positive or negative infinity, depending on
        # the sign of y
        z = float('inf') * y / abs(y)
    else:
        z = float(y) / float(x)
    theta = math.atan(z)
    # Convert angle to degrees, as default math.atan is radians
    theta *= 180.0 / math.pi
    # Now, we only have the angle
    return r, theta


# Should get r = 1, t = 90
x, y = 0, 1
r, t = cartesian_to_polar(x, y)

# NOTE, we can print the results in many ways!
print("Converted (%.2f, %.2f) to (r, theta) = (%.2f, %.2f)" % (x, y, r, t))
print("Converted (" + str(x) + ", " + str(y) +
      ") to (r, theta) = (" + str(r) + ", " + str(t) + ")")
print("Converted (", x, ", ", y, ") to (r, theta) = (", r, ", ", t, ")")
```

Typically, we use a text editor to write Python scripts like these. Windows and macOS have their own default text editors that come pre-installed, but there are also other text editors that have more features that are useful for programming (we personally recommend Sublime Text). If we take all the text in the Python script above and try to save it, the default option when we try to save this file will be to save it as a text file (.txt). To properly save it as a Python script, we just need to change the file extension from ".txt" to ".py". This can be done by simply typing .py at the end of your filename instead of .txt. For example, we can save the script above as "cart_to_pol.py" in our text editor and then run it in the terminal as such:

```
~ python cart_to_pol.py
Converted (0.00, 1.00) to (r, theta) = (1.00, 90.00)
Converted (0, 1) to (r, theta) = (1.0, 90.0)
Converted ( 0 ,  1 ) to (r, theta) = ( 1.0 ,  90.0 )
```

If you are using a previous version of Python before Python 3, then you may be seeing something different than what is shown above. If you are interested, some specific differences between Python 2 and Python 3 can be found here.

Variables

Now, let's go over a quick review of variables in python. We can assign a variable simply by equating some name to a value. For example:

```
# This is a declaration of some variables
a1 = 3      # An integer, or "int" for short
a2 = 3.0    # A float
a3 = 3E-4   # Also a float

# Note, some numbers, once very large, will take on a different data type.
# We will ignore this for right now, but keep it in mind.

b = "This is some string"  # A string, or "str" for short
c = [1, 2, 3, 4, 5]      # A list of integers
d = range(1, 6)        # A range of integers

e = True  # A boolean, that is, a variable holding either True or False

# This one will look weird, but it is checking if c == d, and storing the
# boolean (True in this case) in the variable f
f = c == d

g = (1, 2, 3)  # A tuple. Essentially a list, but you cannot
               # add/remove elements without redefining it.
```

Further, we can do basic arithmetic simply by typing in our equation! Note, python holds the standard +, -, *, and / symbols; however, to do exponents you must use **. For example:

```
# Declare our variables

# What data types are these?
a = 4
b = 5

c = a / b

print(c)
```

If we ran the above code in Python 2, we would get 0. But, that doesn't make sense does it? That's because previous versions of Python would assume data types unless you are specifically telling it what to do. In this case, a and b are both integers, so it would try to find the integer value associated with 4/5. As $4/5 = 0.8$, it drops (the technical term being floor) it to 0. To get the correct answer, we would need to *cast* one of these variables to a float, and Python will realize the solution is no longer constrained to the set of integers. When using Python 3, we do not see this issue and we should get 0.8 regardless of whether we cast the variables to floats or not. However, it is still important to keep data types in mind, as other problems can always arise in the future.

```
a = 4
b = 5

c1 = a/b
c2 = float(a) / b
c3 = a / float(b)
```

In the above case (for Python 2), c1 is 0, but $c2 = c3 = 0.8$. If we are using Python 3, the variables c1, c2, and c3 should all be equal to 0.8. For more information on variables and data types, there are a few resources linked here: basics of Python variables and Python 3 documentation for built-in data types. The documentation contains a lot of information that may be a bit overwhelming. Making use of your Internet browser's find function will be useful for if you want to find information on a specific keyword or topic that you are looking for.

Functions

Sometimes we write a piece of code that is very useful, and we might want to use it multiple times. For example, say we want to find the sign of a number (is it positive, negative, or 0). We can do so as follows:

```
num = 3.0
sign = num / abs(num)
```

Note, we've used a function called "abs()" which takes a value, and returns the absolute value of it. This is a default function in python. Unfortunately, "sign()" is not. So let's make it a function:

```
# Start a function
def sign(num):
    if num == 0:
        return 0
    else:
        return num / abs(num)

# As the next line of code is no longer indented, the function has ended here.

s = sign(3.0)
```

So, we've done a few things here. The first thing is that we've used the *def* keyword to start a function. This simply tells Python: "hey, the next thing you see is my function name and any parameters it needs." Afterwards, we put a colon and then the code for our function. Further, you'll notice the code has been INDENTED! Python needs to know what code, following the line containing def, is in the function. To do this, it takes into account whitespace (spaces, tabs, newlines). So we can indent in 4 spaces, a tab, 2 spaces, whatever you want, JUST BE CONSISTENT! A common error is to have some things tabbed and others indented with spaces. Many advanced text editors, such as Sublime Text, can handle this automatically. Afterwards, we have our code, and the *return* keyword. "Return" will end the function, and give back the data on the right of the keyword. In the above code, we either return 0, for when the number has no sign, or ± 1, depending on the sign of num.

Functions are POWERFUL tools, allowing us to be lazy programmers (the best kind!). We can re-use code we know works well. Further, it allows us to use other people's code! Remember when we called "import math" previously? We were calling the math code that someone else had already written! That is, somewhere else in the computer is a file with the name "math.py", and in it are functions like "sqrt()", "cos()", and more. Using "math.sqrt()" we are able to call the "sqrt()" function from the "math.py" file.

Conditionals

To program there will be many situations in which you need code to act differently depending on a set of inputs. This is where conditionals come into play. A conditional in python is easy, as it reads like english. Take the following code for an example - we are trying to simulate catching a Pokemon:

```python
import random

# Some abstract function that catches the Pokemon
def throw_ball(name):
    ...

# Define boolean variable for if we encounter a pokemon or not
pokemon_encountered = random.random() < 0.5

# Define boolean variable for if the encountered pokemon is Mew
pokemon_is_mew = random.random() < 0.001

# Define various boolean variables based on what kinds of Poke balls we have to use
have_pokeball = True
have_greatball = False
have_ultraball = False
have_masterball = True

# Here are our conditional statements:
if pokemon_encountered:
    if pokemon_is_mew:
        if have_masterball:
            throw_ball("master")
        elif have_ultraball:
            throw_ball("ultra")
        elif have_greatball:
            throw_ball("great")
        elif have_pokeball:
            throw_ball("poke")
        else:
            print("NOOOOOOOOO!!!")
    else:
        if have_pokeball:
            throw_ball("poke")
        else:
            print("Oh well...")
```

The keywords to focus on are highlighted in blue: *if*, *else*, and *elif*. It is good practice to write conditional statements in a way that is readable and easy to understand.

Loops

What would we do if we came across the following code?

```
numbers = list(range(100))
scale = 4.0

numbers = numbers / scale
```

The goal here is to take the numbers 0, 1, 2, ..., 99, and divide them all by a scaling value. Unfortunately, this throws an error when we try to run the code! Let's try doing this:

```
numbers = list(range(100))
scale = 4.0

numbers[0] = numbers[0] / scale
numbers[1] = numbers[1] / scale
numbers[2] = numbers[2] / scale
numbers[3] = numbers[3] / scale
numbers[4] = numbers[4] / scale
numbers[5] = numbers[5] / scale
numbers[6] = numbers[6] / scale
```

This approach doesn't throw any errors when we try to run the code. But it is definitely very tedious to write this out for the 100 numbers that we are trying to modify. Here, we can make use of loops to *abstract* away this boring work! When we use loops, our code could read as follows:

```
numbers = list(range(100))
scale = 4.0

for i in range(len(numbers)):
    numbers[i] = numbers[i] / scale
```

Further, if we don't want to use the range function, Python has a handy function called *enumerate* that makes it easier to loop through a list:

```
numbers = list(range(100))
scale = 4.0

for i, num in enumerate(numbers):
    numbers[i] = num / scale
```

Essentially, using enumerate in this for loop will return the index, i, and the value, num, associated with numbers[i]. Being in a for loop, it loops through all these iteratively. That is, we get:

```
i, num = 0, 0
i, num = 1, 1
i, num = 2, 2
...
i, num = 99, 99
```

Albeit boring in this situation in which i == num for all instances, if we were looping through a different list we might find this useful! This type of loop is called a *for* loop, which you can see from our use of the keyword *for*. What if we have an example where we don't know how large our loop is? What if, all we know is that we would loop as long as some function keeps telling us to loop. Here, we can use a *while* loop as follows:

```
while func_says_loop():
    # Do some code here
    print("Hello... I'm waiting... please let me stop...")
```

Cheat Sheet

So, what have we gone over so far?

1. Variables
   (a) Integers, cast using “int(x)”, hold numbers like -1, 3, 0, 434
   (b) Floats, cast using “float(x)”, hold numbers like 3.412, -23.1, 3.0E-3
   (c) Strings, cast using “str(x)”, hold words/sentences. They need to be specified using qutoations (single or double, but be consistent) ex. ”string1”, ’string2’
   (d) Lists, cast using “list(x)”, hold a sequence of variables. ex. [1, 2, 3, 4, ’a’, ’b’, ’c’], [’e’]
   (e) Tuples, cast using “tuple(x)”, hold a sequence or variables. ex. (1, 2, 3), (’t’, 3, ’x’)
   (f) Booleans, cast using “bool(x)”, holds a logical statement of either True or False.

2. Functions
   (a)

```
# An example function
# Uses def keyword to start the function
def sum(a, b, c=0):
    # Note, we MUST pass the function a and b, but c if not
    # passed is assumed to be 0.
    value = a + b + c
    # We can use the return keyword to return the result to
    # the section of code calling the function
    return value

# Calls the sum function and returns the total to the v variable here.
v = sum(3, 2, 1)
```

   (b)

```
# We can get functions that someone else has written using import
import math

# To call functions from math, we use math + . + function name
# ex. math.cos(), math.sin()
# We also can get variables from that file: math.pi
v = math.cos(1)
```

3. Conditionals

```
a = 1
b = 10
if a > b:
    print("This will never print as 1 is not greater than 10")
elif a < 0:
    print("Neither will this, as 1 is not less than 0")
else:
    print("This is the default case, runs if nothing else ran.")
```

4. Loops

(a)

```python
# This will call a function 10 times
for i in range(10):
    call_some_random_function()  # This is a made up function for demo
```

(b)

```python
# This will loop through the values of a list
s_list = ['H', 'e', 'l', 'l', 'o']
s_msg = ''
for s in s_list:
    s_msg += s

print(s_msg)
```

(c)

```python
# This will loop through the values of a list, as well as the corresponding
# indices
s_list = ['H', 'e', 'l', 'l', 'o']
s_msg = ''
for i, s in enumerate(s_list):
    print("Currently on letter %d of s_list" % i)
    s_msg += s

print(s_msg)
```

(d)

```python
# This will loop through the values of a list, until
# some specific one is found, in which we break out of the loop
s_list = ['H', 'e', 'l', 'l', 'o', 'STOP']
s_msg = ''
i = 0
# We make an infinite loop here
while True:
    s = s_list[i]
    if s == "STOP":
        # Break keyword will ‘jump’ out of the current loop. That is, the
        # code will continue from right after this while loop.
        break
    else:
        s_msg += s
        i += 1
        # continue keyword will skip the rest of this loop, but not break
        # out of it.
        continue
    print("You will never see this printed to the terminal")

print(s_msg)
```